COMPUTER
APPLICATIONS
UNLIMITED.

M-ZAL™ RELEASE 2 MANUAL

by

Jeff Krantz

and

David Willen

M-ZAL RELEASE 2

M-ZAL RELEASE 2

I. INTRODUCTION

Release 2 of the M-ZAL development system contains many enhancements and extensions to the original M-ZAL. The authors wish to thank the many M-ZAL users who wrote or called with suggestions for some of these new features.

This manual describes and explains all of the new features of M-ZAL RELEASE 2 and is meant to be used in conjunction with the M-ZAL SYSTEM MANUAL. The changes made by release 2 can be summarized as follows:

1) A new utility program, LEXCONV, has been added to facilitate conversion between the various source file formats used by TRS-80 assemblers.
2) New features for the TXEDIT (text editor) program:
   a) Warm Start Option.
   b) Show Free Space command (CLEAR ?).
3) New features for the ASMBLR (assembler) program:
   a) Pseudo-ops DEFB, DEFW, and DEFS can now be abbreviated as DB, DW, and DS.
   a) Conditional Assembly.
   b) Recursive Macro Language.
4) New features for the LINKER (module linker) program:
   a) DISPLAY command.
   b) ZAP command (module superzap capability).
   c) SPEED command (set tape speed, Mod 3 only).

M-ZAL RELEASE 2 is distributed on a non-system disk for either the TRS-80 Model I or Model III. You must therefore start out by placing your own system disk into drive 0 and the M-ZAL disk into drive 1. The release 2 disk contains all of the files as defined on page 3 of the M-ZAL SYSTEM MANUAL. Only the following files have been changed from release 1:

TXEDIT/CMD TXEDIT/RLD ASMBLR/CMD LINKER/CMD

Only one new file has been added, it is LEXCONV/CMD and it contains the new lexical converter program.

The rest of this manual is dedicated to explaining the new features of each M-ZAL program. An understanding of the original M-ZAL system, as described in the M-ZAL SYSTEM MANUAL, is assumed.

II. LEXCONV: LEXICAL CONVERTER PROGRAM

There are a number of editor/assembler products on the market now and several different source file formats are in common use. The LEXCONV program allows you to convert disk source files from one format to another. In addition, LEXCONV will optionally check to insure that the source file contains acceptable syntax for the M-ZAL assembler, and will make adjustments as needed.

1) File Formats Supported

LEXCONV will allow you to convert disk files between any of the following four formats:

1) M-ZAL	- The file format used by the M-ZAL programs TXEDIT, ASMBLR, and LINKER.
2) APPARAT EDTASM	- The file format used by Apparat's extension to Radio Shack's EDTASM. This format is also used by EDAS.
3) MACRO-80	- The file format used by Microsoft's MACRO-80 assembler sold by Radio Shack.
4) UNNUMBERED ASCII	- This file format is used by some word processor programs and is also supported by the LIST command of most Disk Operating Systems (DOS). It does not include line numbers on each source line.

A detailed definition of the M-ZAL source file format will be found in the M-ZAL SYSTEM MANUAL on page 63. A definition of the other file formats can be found in the book "TRS-80 Disk and Other Mysteries" by H. C. Pennington.

The M-ZAL file format is the most space efficient of those formats that support numbered lines. This means that if you store your source programs in this format they will take up less disk space than if you stored them in one of the other formats. The difference is not significant for short source programs but can become quite substantial as your programs grow in size.

2) How to use LEXCONV

LEXCONV is invoked from DOS READY mode by simply typing in the command LEXCONV and hitting ENTER. This program is self documenting and explains all of your options as it prompts you for information.

You will be asked for a source filename. This is the name of the disk file containing the program that you wish to convert. You will then be asked for the source file type. This tells LEXCONV what format the source file is in. You must reply to this prompt with a number from 1 to 4, corresponding to the four different file types described on the preceding page (your choices are described on the screen by LEXCONV as well).

If you discover that you have accidently given an incorrect reply to one of the prompts, you can hit the CLEAR key. This will cause the LEXCONV program to restart and let you correct your error. Hitting the CLEAR key in response to the first prompt will return you to DOS READY mode.

Once you have specified the source file to be converted, you will be asked for the target filename. This is the name of the file that LEXCONV will place the converted program into. The target file can be the same as the source file because LEXCONV reads the entire source file into memory, converts it as needed, and then writes it back out to the target file. This means that you can convert large files from one format to another without having to worry about running out of disk space (simply specify the same name for both source and target filenames, it is a good idea to have a backup of the source file before doing this).

LEXCONV will now ask you for the target file format. This is the format that LEXCONV will convert your program into. Your choices are the same as for the source file format.

LEXCONV will now ask you two more questions. You will be asked if you wish to compress blanks. The reply can be Y for yes or N for no. Under most circumstances you should reply yes which will cause multiple blanks to be converted into horizontal tab characters (09H) where possible. We have included this option so that you can create UNNUMBERED ASCII format files for use with word processing programs that do not support horizontal tab characters. Note that most files will be significantly larger if converted without the compress blanks option.

Finally, you will be asked if you wish to perform lexical conversion. The reply to this question is Y for yes or N for no. If you reply yes, LEXCONV will perform additional

processing for each line of the program being converted. This optional processing has been designed to detect certain assembler language syntax forms that are not supported by the M-ZAL assembler, but which may appear in programs written for use with other assemblers. LEXCONV will attempt to convert such statements to their M-ZAL equivalents. The specific conditions handled by this option are described in the next section of this chapter.

After the last question is answered, LEXCONV will begin processing. It will open the source file and read the program contained therein, assuming it is in the format that was specified. As the program is read into memory, blank compression and/or lexical conversion will be performed if requested. When the entire source file has been read it will be closed and the target file will be opened. The program will then be written out to the target file in the format specified. If an i/o error or out of memory condition is encountered, you will be given the option of continuing or terminating (i/o error codes are given in decimal).

When LEXCONV has completed the requested conversion you can hit the ENTER key to refresh the menu and specify another conversion or hit the CLEAR key to return to DOS.

As you can well imagine, LEXCONV is a very versatile program. The following sample LEXCONV session will demonstrate only one of LEXCONV's many uses.

Suppose you have a program written for use with MACRO-80 and you wish to assemble it using the M-ZAL assembler. The program is stored on your system disk as file "PGM5/ASM" . You want to convert it for use with M-ZAL and place the new copy on the disk in drive 1 with a filename of "PGM5A/ASM" . Your LEXCONV session would look like this:

```
ENTER SOURCE FILENAME ==> PGM5/ASM:0
ENTER SOURCE FILE TYPE (1-4) ==> 3

ENTER TARGET FILENAME ==> PGM5A/ASM:1
ENTER TARGET FILE TYPE (1-4) ==> 1

DO YOU WISH TO COMPRESS BLANKS WHERE POSSIBLE (Y/N) ? Y
DO YOU WISH TO PERFORM LEXICAL CONVERSION (Y/N) ? Y
```

3) The Lexical Conversion Option

As mentioned earlier, the lexical conversion option gives you the ability to automatically modify certain assembler language statements so that they will be acceptable to the M-ZAL assembler. This chapter defines exactly what conditions are handled by the converter.

- The M-ZAL assembler requires that all comment lines begin with a semicolon in column one. Some assemblers accept all blank lines as comments (M-ZAL will give a "NO OPCODE" error). Some assemblers allow comment lines to begin after column 1 with a semicolon (M-ZAL will give an "INVALID OPCODE" error). The lexical converter option will place a semicolon in column one of all blank lines. It will also place a semicolon in column one if it detects a semicolon on a line prior to any other non-blank character.

- Some assemblers allow labels to be followed by colons. Some assemblers require labels to be followed by colons. The M-ZAL assembler does not accept colons on labels. The lexical converter option will detect a colon if it appears immediately after a valid label and will remove it. Thus the statement

```
GETPARM: LD      HL,(PARM)
```

  will be converted into

```
GETPARM  LD      HL,(PARM)
```

- Some assemblers allow labels to be equated to the current location counter by simply following them with a colon and leaving the rest of the line blank. The lexical converter option will detect this condition and convert it to the equivalent M-ZAL statement. Thus the statement

```
XTABLE:
```

  will be converted into

```
XTABLE  EQU     $
```

- An additional feature of the lexical converter option is that it will always remove trailing blanks from every line it processes. This helps to reduce the amount of disk space required by any assembler program.

III. NEW FEATURES IN TXEDIT (TEXT EDITOR PROGRAM)

Two new features have been added to the TXEDIT (text editor) program. They are the warm start option and a new command to display free space (CLEAR ?).

1) Warm Start Option

The text editor can now be invoked and instructed to use the contents of the text buffer already in memory. This is very useful when you accidently hit D (return to DOS) from the exit menu instead of S (save file). In such a case the text that you had been editing is still in memory and can be retrieved by simply typing in TXEDIT and then replying to the ENTER FILENAME prompt with a plus sign character ("+").

Note that if the contents of memory above 5CC0H have been disturbed, this option will not work as the text buffer will have been destroyed. In this event you may experience very erratic operation of the text editor and will probably have to re-boot your system.

You can use this option to return to DOS, perform another function, and return to the text editor with your text intact. Remember this is possible only if you do not alter memory above 5CC0H while back in DOS. Unfortunately, the DIR command under TRSDOS does destroy the text buffer. If you are using a more sophisticated operating system such as DOSPLUS, however, you can now exit from the editor to perform a DIR and then return to the editor with your text as you left it.

2) Display Free Space Command

The text editor now has a new command which displays how much free space you have left in your text buffer. To use this new command hit the CLEAR key while in normal edit mode and then hit the ? key. A message will appear briefly to indicate how many bytes are available in the text buffer.

IV. NEW FEATURES IN ASMBLR (ASSEMBLER PROGRAM)

The most significant change in M-ZAL RELEASE 2 is the addition of conditional assembly and macro capabilities. These extensions to M-ZAL were designed to provide powerful pre-processing facilities for the sophisticated assembler language programmer. The language extensions are not intended to be compatible with other Z-80 macro assemblers as no standard language definition exists.

1) Conditional Assembly

Conditional assembly refers to the ability to decide dynamically, at assembly time, whether or not a section of code is to be assembled. Conditional assembly is supported in M-ZAL by two new pseudo-ops, IF and ENDIF.

The format of the IF statement is as follows:

```
IF      value operand
```

The operand of the IF statement may be any valid expression. When the assembler encounters the IF statement, it evaluates the expression. If the expression is not equal to zero then assembly continues normally. If the expression evaluates to zero then the assembler stops assembling statements until it reaches the next ENDIF statement.

The format of the ENDIF statement is as follows:

```
ENDIF
```

Note that the ENDIF statement has no operands. Note also that labels are not permitted on either the IF or the ENDIF statements as they do not generate object code.

Example: suppose you want your program to display a prompting message while it is being developed, but eventually you want to remove this message from the program. Write your program as follows:

```
START   CALL    1C9H    ;CLEAR SCREEN
        IF      1       ;ASSEMBLE THE NEXT
;                        TWO STATEMENTS:
        LD      HL,MSG1
        CALL    21BH    ;DISPLAY MESSAGE
        ENDIF
        LD      HL,IB   ;INPUT BUFFER
        CALL    40H     ;READ REPLY
        ...     ...     rest of your program
MSG1    DEFM    'ENTER YOUR ANSWER'
        END     START
```

Then when it is finished and you wish to deactivate the code that displays the prompting message (without actually deleting it so you can reinstate it if needed), make the following change:

```
START   CALL    1C9H    ;CLEAR SCREEN
        IF      0       ;DONT ASSEMBLE THE
;                        NEXT TWO STATEMENTS:
        LD      HL,MSG1
        CALL    21BH    ;DISPLAY MESSAGE
        ENDIF
        LD      HL,IB   ;INPUT BUFFER
        CALL    40H     ;READ REPLY
        ...     ...     rest of your program
MSG1    DEFM    'ENTER YOUR ANSWER'
        END     START
```

The conditional assembly feature has many more uses than the example just shown. For example, let us assume that you are writing a very large program which is going to just barely fit into memory. If the main body of the program ever gets as large as 8K then you want to automatically include a special routine which manages your buffer space under very tight memory constraints. This can be accomplished as follows:

```
START   ...             ;START OF PROGRAM
        ...
        ...             ;MAIN BODY OF PROGRAM
        ...
PGMSIZE EQU     $-START ;SIZE OF MAIN BODY
        IF      PGMSIZE<-13  ;IF >= 8K THEN
;                            CONTINUE ASSEMBLING.
        ...
        ...             ;SPECIAL ROUTINE
        ...
        ENDIF
        END     START
```

2) Nested Conditional Assembly

Note that every IF statement in a program must have a matching ENDIF statement or an error message will be given (see new error messages chapter). It is also possible to nest IF and ENDIF statements. As long as the operand of each IF statement is non-zero, assembly will continue. Once an IF statement is encountered with a zero operand, statement assembly will stop and will not resume until the matching ENDIF statement is encountered, even if a more deeply nested IF statement does have a non-zero operand. There is no limit to how deeply IF and ENDIF statements can be nested! Examples follow.

```
START   ...                 ;START OF PROGRAM
        ...
        IF      0           ;STOP ASSEMBLY
        ...
        ...
        IF      1           ;STILL NO ASSEMBLY
        ...
        ...
        ENDIF
        ...
        ...
        ENDIF
        ...                 ;ASSEMBLY RESUMES
        ...
        END     START
```

```
START   ...                 ;START OF PROGRAM
        ...
        IF      1           ;ASSEMBLY CONTINUES
        ...
        ...
        IF      0           ;STOP ASSEMBLY
        ...
        ...
        ENDIF
        ...                 ;ASSEMBLY RESUMES
        ...
        ENDIF
        ...
        ...
        END     START
```

```
START   ...                 ;START OF PROGRAM
        ...
        IF      1           ;ASSEMBLY CONTINUES
        ...
        ...
        IF      1           ;ASSEMBLY CONTINUES
        ...
        ...
        ENDIF
        ...
        ...
        ENDIF
        ...
        ...
        END     START
```

```
START   ...             ;START OF PROGRAM
        ...
        IF      0       ;STOP ASSEMBLY
        ...
        ...
        IF      0       ;STILL NO ASSEMBLY
        ...
        ...
        ENDIF
        ...
        ...
        ENDIF
        ...             ;ASSEMBLY RESUMES
        ...
        END     START
```

3) IF Statement Restrictions

Because of M-ZAL's multi-pass nature, certain restrictions must be placed on the use of IF statements. Specifically, all IF statement operands must be defined at the time they are evaluated or else a terminal error will occur. This error will occur during the assembler's first pass so you will only see the offending IF statement and the statement which follows it (the listing pass is not yet being made). For example:

```
00001  START   ...             ;START OF PROGRAM
00002          ...
00003          IF      ALPHA   ;ALPHA NOT YET DEFINED
00004          ...
00005          ...
00006          ENDIF
00007          ...
00008  ALPHA   EQU     5       ;DEFINE ALPHA (TOO LATE!)
00009          ...
00010          END     START
```

In the above example, the operand of the IF statement is not defined when it is encountered. Assembly will be terminated and statements 3 and 4 will be listed to indicate where the problem is located. Moving statement 8, which defines the label ALPHA, to before the IF statement will solve the problem.

Note that the $ (location counter) operand is considered undefined when it depends upon an as yet undefined ORG statement, thus in the following example the IF statement is also invalid:

```
START   ...
        ORG     ALPHA
        ...
        IF      $-28    ; $ NOT YET DEFINED
        ...
ALPHA   EQU     10      ; DEFINES PREVIOUS ORG
```

4) Introduction to Macros

Macros are most commonly used to simplify programs when several occurrences of similar pieces of code need to be generated. In this application, the programmer usually writes a macro which defines a prototype (or skeleton) of a sequence of assembler language statements. This is called the macro definition and is usually placed at the beginning of the entire program. Subsequently, at the various points in the program where this sequence of instructions is desired, the programmer places a statement which contains the macro name as an opcode. This statement is called a macro invocation. When the assembler encounters the macro invocation, it replaces it with the sequence of statements defined in the macro definition. This process is called macro expansion.

As a very simple example, let us assume that you are writing a program which requires that you define, at several different points, a byte containing the value 1 followed by a word containing the value 3C00H. Instead of typing in the two line sequence:

```
        DEFB    1
        DEFW    3C00H
```

at every point where it is desired, you can define a macro, named MAC1 , by placing the following four statements at the beginning of your program:

```
MAC1    MACRO           ;DEFINE MACRO
        DEFB    1
        DEFW    3C00H
        ENDM            ;END MACRO DEFINITION
```

Once you have done this, you can cause the two statements desired to appear anywhere else in the program by simply entering a line containing the macro name, thus:

```
        LD      HL,VALUE1
        RET
VALUE1  MAC1
```

will cause the sequence:

```
        LD      HL,VALUE1
        RET
VALUE1  DEFB    1
        DEFW    3C00H
```

to be assembled.

The power of macros is made much greater by a feature known as symbolic substitution. This allows you to customize each macro invocation with variable data. The variable information is specified as operands on the macro invocation statement. Each operand corresponds to a symbolic parameter on the macro definition statement. As the macro is expanded, each occurrence of the symbolic parameter is replaced with the corresponding operand.

Symbolic parameters in M-ZAL always begin with a question mark (?). As a simple example, suppose you are writing a program which has to increment the contents of a word in memory at several different points. You could write the following macro definition and place it at the beginning of your program:

```
INCWORD MACRO   ?WORD          ;MACRO DEFINITION
        PUSH    HL             ;SAVE HL REG
        LD      HL,(?WORD)     ;GET VALUE OF WORD
        INC     HL             ;INCREMENT IT
        LD      (?WORD),HL     ;UPDATE WORD
        POP     HL             ;RESTORE HL
        ENDM                ;END MACRO DEFINITION
```

Now when you place the statement:

```
INCWORD CHARLIE
```

in your program, it will cause the following sequence to be expanded:

```
PUSH    HL             ;SAVE HL REG
LD      HL,(CHARLIE)   ;GET VALUE OF WORD
INC     HL             ;INCREMENT IT
LD      (CHARLIE),HL   ;UPDATE WORD
POP     HL             ;RESTORE HL
```

Observe how each occurrence of the symbolic parameter ?WORD in the macro definition is replaced with its corresponding operand when the macro is expanded (in this case the operand on the INCWORD macro invocation statement is CHARLIE). Note that you have effectively defined a new opcode called INCWORD which provides a 16 bit memory increment!

5) Macro Definitions

You may define as many macros as you wish in your program but all macro definitions must appear at the beginning of the program, before any statements that generate, or affect the generation of object code. One excellent approach is to place all of your macros in a seperate source file and use a *INCL statement at the beginning of the program to include them. A file containing a set of macros used in this manner is referred to as a macro library.

The only statements that are allowed before or in between macro definitions are comments and assembler command statements. Assembler command statements are those that begin with a * , such as *LIST ON. If a macro definition appears after any assembler language statements (such as ORG, LD, DEFB, EQU, CALL, EXTRN, etc.) then an "ILLEGAL MACRO DEFINITION" error will occur.

A macro definition is a sequence of statements that always begins with a MACRO statement and ends with an ENDM statement. The statements that appear between the MACRO and ENDM statements comprise the body of the macro and are called the prototype statements.

The format of the MACRO statement is as follows:

```
label   MACRO   {symbolic parameters}
```

A label is always required on a MACRO statement. The label defines the name of the macro; this is the name by which the macro will be invoked later on. The label may be any valid label as defined in Chapt. IV.1 of the M-ZAL SYSTEM MANUAL (ASSEMBLER SECTION).

No conflict will occur if the same name is used later on in the program as a label on an assembler language statement that generates object code.

Zero or more symbolic parameters may be specified on any MACRO statement. If more than one symbolic parameter is specified, they should be seperated by commas. There is no limit to how many symbolic parameters may be specified.

Each symbolic parameter starts with a question mark (?). The rest of the symbolic parameter may be any valid label as defined in Chapt. IV.1 of the M-ZAL SYSTEM MANUAL.

Examples of valid MACRO statements:

```
MULTIPLY MACRO  ?OPERAND1,?OPERAND2
ADD      MACRO
FREDDIE  MACRO  ?U,?V,?W          ;FREDDIE'S MACRO
```

Examples of invalid MACRO statements:

```
BIRD     MACRO   ?A,?B    ;invalid symbolic parm. names
IX       MACRO            ;invalid macro name
TRILL9   MACRO   ?1,?2    ;invalid symbolic parameters
FLUTE    MACRO   ?UV?W    ;invalid symbolic parameter
```

The MACRO statement is followed by one or more prototype statements which will be inserted into the program when the macro is expanded. Prototype statements can be any kind of statement desired, they are not checked for validity or

executed until the macro is expanded. Thus a *LIST OFF statement within a macro definition will not suppress the listing of the rest of the macro but will take effect later when the macro is invoked. (You can place *LIST OFF statements before MACRO statements to suppress listing of macro definitions if desired.)

The only exception to this rule is the *INCL statement. If a *INCL statement is placed within a macro definition it will be executed immediately and the INCLuded text will be added to the macro definition. The *INCL statement will not be executed when the macro is invoked.

Prototype statements may contain references to the symbolic parameters that were defined on the MACRO statement. These references will be replaced with the corresponding operand of the invoking statement when the macro is expanded.

Symbolic parameter references may be placed anywhere on any prototype statement. They may appear in the label, opcode, operand, and/or comment fields of prototype statements. They may also appear within quoted character strings on prototype statements.

Symbolic parameters may be combined with other characters on prototype statements by means of the concatenation operator, which is the period character (.). For example, consider the macro definition:

```
COMB    MACRO   ?VARTEXT
        DEFM    '?VARTEXT.XYZ'
        ENDM
```

The macro invocation:

```
        COMB    ABC
```

will expand into:

```
        DEFM    'ABCXYZ'
```

The macro invocation:

```
        COMB    'ABC DEF '
```

will expand into:

```
        DEFM    'ABC DEF XYZ'
```

The macro invocation:

```
        COMB
```

will expand into:

```
        DEFM    'XYZ'
```

Note that the concatenation operator (.) is not required under certain conditions. Concatenation is implied whenever a symbolic parameter is followed immediately by

- a blank
- a comma (,)
- any operator (+, -, etc.)
- a semicolon (;)
- a right parenthesis (")")

Regardless of this fact, whenever a symbolic parameter is followed by a period (.), concatenation is implied. If you wish the macro expansion to actually contain the period character after a symbolic parameter, you must follow the symbolic parameter with two periods (..), for example:

```
FRACTION MACRO    ?X
         DEFM     '?X..15'
         ENDM
```

The macro invocation:

```
         FRACTION 10
```

will expand into:

```
         DEFM     '10.15'
```

6) Special Symbolic Parameters ?INDEX and ?PARM#

Any macro definition may reference the special symbolic parameters ?INDEX and ?PARM#. These parameters are defined automatically for every macro and have special significance.

The ?INDEX symbolic parameter will be replaced when the macro is expanded by a four character value which will range from "0000" to "9999". This value starts at zero and is incremented by one after each macro invocation in the program. ?INDEX therefore provides a value which is unique for each macro expansion. When writing macros that must generate internal address labels, ?INDEX can be used to insure that multiple invocations do not cause MULTIPLE DEFINED LABEL errors.

For example, consider the following macro which subtracts a parameterized value from the A reg and then checks to see if the result is negative. If it is, then A is loaded with another parameterized value:

```
SUBCK    MACRO    ?P1,?P2
         SUB      ?P1
         JR       NC,GL?INDEX
         LD       A,?P2
GL?INDEX EQU      $
         ENDM
```

If this macro is invoked several times as follows:

```
        SUBCK   10,20
        SUBCK   50,75
```

then the resulting macro expansions will appear as:

```
        SUB     10
        JR      NC,GL0000
        LD      A,20
GL0000  EQU     $
        SUB     50
        JR      NC,GL0001
        LD      A,75
GL0001  EQU     $
```

Observe how each macro expansion used a different value for ?INDEX and thus created a unique address label for the jump instruction target. This is only one of the many ways in which ?INDEX can be used.

The ?PARM# symbolic parameter will be replaced during macro expansion with a two character value which will range from "00" to "99". This value will represent the number of operands that were specified on the invoking statement.

The following example illustrates the operation of ?PARM#:

```
COUNT   MACRO   ?U,?V,?W
        DEFB    ?PARM#
        ENDM
```

The statement:

```
        COUNT
```

will generate:

```
        DEFB    00
```

The statement:

```
        COUNT   A,B
```

will generate:

```
        DEFB    02
```

7) Detecting Null Macro Operands (! operator)

From the preceding example you may have noticed that it is not necessary to provide a matching operand for each symbolic parameter in a macro. If no value is specified to correspond to a symbolic parameter when a macro is invoked, then that parameter will be considered "null" during the macro expansion. If you turn back to page 14 you will see another example of this at the bottom of the page (the COMB macro example).

When writing a macro it is often useful to be able to tell whether or not the operand being passed to a symbolic parameter is null. This can be accomplished by prefixing the symbolic parameter with the exclamation point (!) operator.

When the macro is expanded, this operator will cause the parameter to be replaced with a "0" if the corresponding operand is null, or with a "1" if the corresponding operand is not null. This result can be used as the operand of an IF statement to conditionally expand part of the macro body.

For example, consider a macro which is passed a string of characters and an optional number. The object of the macro is to assemble the string of characters into memory and also assemble a word containing the number if it is present.

```
GENMSG  MACRO   ?STRING,?NUMBER
        DEFM    '?STRING.'
        IF      !?NUMBER
        DEFW    ?NUMBER
        ENDIF
        ENDM
```

The statement:

```
        GENMSG  TWENTY,20
```

will generate:

```
        DEFM    'TWENTY'
        IF      1
        DEFW    20
        ENDIF
```

The statement:

```
        GENMSG  HUNDRED
```

will generate:

```
        DEFM    'HUNDRED'
        IF      0
        ENDIF
```

It is often desirable to force a null operand on a macro invocation statement. This can be easily accomplished by placing two consectutive commas in the operand list, or by placing a leading or trailing comma in the list. If, for example, a macro is defined with the statement:

```
MULTI   MACRO   ?U,?V,?W,?X
```

then the macro invocation statement:

```
        MULTI   1,,2,3
```

will expand with a null value for ?V. Values of 1, 2, and 3 will be assigned to ?U, ?W, and ?X respectively. Note that the value of ?PARM# will still be 4 since embedded null parameters are counted when ?PARM# is calculated.

Similarily, the macro invocation statement:

```
MULTI   ,2,3,4
```

will expand with a null value for ?U and values of 2, 3, and 4 for ?V, ?W, and ?X respectively. The value of ?PARM# will be "04" because we have actually specified four operands, it just happens that one of them is null. The same technique can be applied to the last symbolic parameter by coding a trailing comma on the macro invocation statement.

8) Macro Comment Statements (.)

Often it is desirable to add comments to macro definitions that we do not wish to see when the macro is expanded. If a normal comment statement (one that starts with a semicolon on column 1) is placed in a macro definition then it will appear when the macro is expanded. In fact, any symbolic parameters on such a comment statement will be replaced by the appropriate operands. To create a comment line that will not appear in macro expansions, start it in column 1 with a period (.). For example, when the macro below is expanded comment statement 3 will appear, but comment statement 2 will not:

```
1 DOCU  MACRO   ?VAL1,?VAL2
2 . THIS MACRO CONTAINS INTERNAL DOCUMENTATION.
3 ; THIS IS THE DOCU MACRO EXPANSION:
4       DEFB    ?VAL1
...     ...     ...
```

9) Terminating Macro Definitions (ENDM statement)

Each macro definition is terminated by an ENDM statement. The format of an ENDM statement is as follows:

```
ENDM
```

Note that a label is not permitted on an ENDM statement. The ENDM statement has no operands.

10) Macro Naming Conventions

As already mentioned, a macro name can be any valid label. You can even redefine standard Z-80 opcodes and M-ZAL pseudo-opcodes by creating macros using their names.

If you define a macro with the name of a Z-80 opcode or M-ZAL pseudo-opcode then you will be unable to use that opcode in your program. For example, if you define a macro named SET then you will be unable to use the Z-80 SET instruction in your program because it would cause the macro to be invoked (you can always generate the desired Z-80 SET instruction via DEFB statements if needed).

The only exceptions to the above are the new M-ZAL pseudo-opcodes MACRO, ENDM, IF, and ENDIF. None of these opcodes can be redefined as macro names, any attempt to do so will result in an "INVALID MACRO NAME" error.

If two macro definitions appear with the same name in one program, the second macro definition will be ignored, no error message will be given. If this macro name is invoked, the first macro definition that used the name will be used for expansion.

All macro definitions are compressed into a special format and stored in memory during the assembly process. The use of macros therefore reduces the amount of memory available for other assembler functions (symbol table, RLD, etc.).

Macro names and symbolic parameter names are not stored in the symbol table and will not appear in the symbol table listing/cross-reference (unless of course they are used elsewhere in the program as regular labels).

11) Macro Invocations

As we have already seen, a macro is invoked by placing its name on an assembler language statement as the opcode. If a label is used on a macro invocation statement, the label will be assigned the value of the current location counter. This means that the label will reflect the address of the first byte of code generated by the macro (unless of course the macro is really strange and contains an ORG statement).

As many operands as are desired may be placed on a macro invocation statement. Operands are seperated from each other by commas and we have already seen how to generate null operands by using extra commas in the operand list. No error message will be given if there are more operands on a macro invocation statement than there are symbolic parameters on the corresponding macro definition.

Note that if no operands are desired on a macro invocation statement then no comment field may appear. This is because the comment would be taken as the first operand.

Any string of characters may be specified as a macro operand. If you wish the string to contain special characters or embedded blanks you must enclosed it in quotes ('). If you wish the operand string to contain the quote character itself, you must specify it as two consective quotes, i.e:

```
COMB    'FREDDIE''S FRIEND'
```

12) Macro Expansions

As we have already explained, when a macro is invoked the contents of the macro definition are "expanded" into the program, with symbolic substitution taking place as needed.

When the assembler lists the program, any statements that are part of a macro expansion will appear with a plus sign (+) character immediately to the right of the line number field. This makes it easy to identify where macros are being used in a program.

The line number field of each statement in a macro expansion will be identical to the line number field of the invoking statement. When looking at a long and complex listing, this feature makes it easy to refer back to the original line of source code which caused a macro expansion. This feature also insures that the symbol table cross-reference will still be usable for locating labels created by macro expansions. We urge our users to keep their source code numbered sequentially by frequent use of TXEDIT's renumber command. If you do this, you will find that it is very easy to locate any error in a program and go right to it in your source code through use of TXEDIT's search command.

If, during macro expansion, a statement is generated which is longer than 128 characters, only the first 128 characters will be used.

If a macro expansion generates a *INCL statement, it will be flagged as an "INVALID OPCODE" error. *INCL statements can only be processed if they are read in directly from a disk source file.

13) Controlling Macro Listing (*MACLIST statements)

You can control the listing of macro expansion statements through use of the new assembler command statements:

```
          *MACLIST OFF
    and   *MACLIST ON
```

These statements may appear anywhere in a program, including within macro definitions. Their action is independent of the action of regular *LIST OFF and *LIST ON statements.

When a *MACLIST OFF statement is encountered it causes the assembler to stop listing any statements that are part of macro expansions. The listing of regular statements continues if it has not already been inhibited by a *LIST OFF statement.

A *MACLIST ON statement will negate the effect of any previous *MACLIST OFF statements. *MACLIST statements can not be nested, the last *MACLIST statement encountered will determine the status of macro listing at any time.

We caution that you can do some very strange things with listing control statements in macros. For example, consider the following:

```
1 FUNNY   MACRO
2 *LIST OFF
3         ENDM
4 ;
5 *MACLIST OFF
6         FUNNY
7 ; THIS STATEMENT WILL NOT BE LISTED.
```

In statement 5, listing of macro expansion statements is turned off. The next statement, number 6, will cause the FUNNY macro to be expanded, but the expansion will not be listed. The listing will therefore not show the *LIST OFF statement in the expansion and there will be no apparent reason why all statements after statement 6 are not listed.

Note that *MACLIST controls the listing of macro expansions. If you want to suppress the listing of macro definitions, use a *LIST OFF statement before the definition itself.

14) Pre-processor Variables (DEFL statement)

You may recall that in the M-ZAL SYSTEM MANUAL we suggested that the DEFL statement not be used since it allows you to dynamically change the value of a label during an assembly. When that manual was written, M-ZAL had no pre-processing capabilities and we felt that the ability to change the value of a label would only get one into trouble. However, this capability is incredibly useful when combined with the conditional assembly and macro facilities of M-ZAL RELEASE 2.

For example, suppose we wish to write a macro that will generate a byte containing a value which increases by one every time the macro is used. We cannot use the ?INDEX special parameter for this task because it is affected by all macro invocations and we wish to keep track of just the invocations of our macro.

The following example illustrates the use of a DEFL variable in accomplishing this task.

```
                1 COUNT    MACRO
                2 COUNTER  DEFL     COUNTER+1
                3          DEFB     COUNTER
                4          ENDM
                5 ;
0005            6 COUNTER  DEFL     5        ;SET INITIAL VALUE - 1
0000            7          COUNT
0006            7+COUNTER  DEFL     COUNTER+1
0000 06         7+         DEFB     COUNTER
0001            8          COUNT
0007            8+COUNTER  DEFL     COUNTER+1
0001 07         8+         DEFB     COUNTER
0002            9          COUNT
0008            9+COUNTER  DEFL     COUNTER+1
0002 08         9+         DEFB     COUNTER
```

Note that the above example is given in the form of a listing as opposed to the previous examples which were given in the form of a list of source statements. The above example contains the macro expansion statements as they would appear after each macro invocation, they can be identified by the + which immediately follows the line number field.

15) Nested and Recursive Macros

Macros may be nested by placing macro invocation statements within the body of macro definitions. Operands may be passed dynamically from one level of macro nesting to another via symbolic parameters. Each macro invocation causes a unique macro expansion to take place, with a unique set of operand values and ?PARM# and ?INDEX values. Note that the perceived value of ?INDEX may appear to go down as well as up during nested macro expansions.

When a macro definition includes an invocation of itself, the macro is said to be recursive. Such macros provide the ability to create "pre-processor loops" which cause the same sequence of statements to be assembled many times.

As an example of a recursive macro, consider the following:

```
TABLE   MACRO   ?VALUE
        DEFB    ?VALUE
        IF      ?VALUE
        TABLE   ?VALUE-1
        ENDIF
        ENDM
```

When called with an operand, this macro assembles a byte containing that value. If the value was zero, then the macro ends. Otherwise, it invokes itself with an operand value that is one less than before. If called with any small positive value, this macro will assemble a sequence of bytes starting with that value and counting down until a byte containing zero is assembled.

For example, the statement:

```
TABLE   5
```

will cause the sequence of bytes 05 04 03 02 01 00 to be assembled, as illustrated below:

```
                  1 ;
                  2 TABLE MACRO   ?VALUE
                  3       DEFB    ?VALUE
                  4       IF      ?VALUE
                  5       TABLE   ?VALUE-1
                  6       ENDIF
                  7       ENDM
                  8 ;
0000              9       TABLE   5
0000 05           9+      DEFB    5
0005              9+      IF      5
0001              9+      TABLE   5-1
0001 04           9+      DEFB    5-1
0004              9+      IF      5-1
0002              9+      TABLE   5-1-1
0002 03           9+      DEFB    5-1-1
0003              9+      IF      5-1-1
0003              9+      TABLE   5-1-1-1
0003 02           9+      DEFB    5-1-1-1
0002              9+      IF      5-1-1-1
0004              9+      TABLE   5-1-1-1-1
0004 01           9+      DEFB    5-1-1-1-1
0001              9+      IF      5-1-1-1-1
0005              9+      TABLE   5-1-1-1-1-1
0005 00           9+      DEFB    5-1-1-1-1-1
0000              9+      IF      5-1-1-1-1-1
                  9+      ENDIF
                  9+      ENDIF
                  9+      ENDIF
                  9+      ENDIF
                  9+      ENDIF
                  9+      ENDIF
                 10 ;
```

From the above example you can see that it is a good idea to keep *MACLIST OFF when using recursive macros. This will help keep your listings readable.

The next page contains another example of a recursive macro. This macro redefines the DEFB pseudo-opcode so that it can be used with up to 10 operands (the DEFB pseudo-opcode as defined in M-ZAL only allows one operand to be specified). If you place this macro in your program you will be able to specify anywhere from 1 to 10 operands on all subsequent DEFB statements. Observe how the exclamation point operator is used to detect when all specified operands have been assembled.

```
                1 DEFB  MACRO   ?E1,?E2,?E3,?E4,?E5,?E6,?E7,?E8,?E9,?E10
                2       IF      !?E1
                3       DB      ?E1
                4       DEFB    ?E2,?E3,?E4,?E5,?E6,?E7,?E8,?E9,?E10
                5       ENDIF
                6       ENDM
                7 ;
0000            8       DEFB    0,1,2,3,4,5,6,7,8,9
0001            8+      IF      1
0000 00         8+      DB      0
0001            8+      DEFB    1,2,3,4,5,6,7,8,9
0001            8+      IF      1
0001 01         8+      DB      1
0002            8+      DEFB    2,3,4,5,6,7,8,9,
0001            8+      IF      1
0002 02         8+      DB      2
0003            8+      DEFB    3,4,5,6,7,8,9,,
0001            8+      IF      1
0003 03         8+      DB      3
0004            8+      DEFB    4,5,6,7,8,9,,,
0001            8+      IF      1
0004 04         8+      DB      4
0005            8+      DEFB    5,6,7,8,9,,,,
0001            8+      IF      1
0005 05         8+      DB      5
0006            8+      DEFB    6,7,8,9,,,,,
0001            8+      IF      1
0006 06         8+      DB      6
0007            8+      DEFB    7,8,9,,,,,,
0001            8+      IF      1
0007 07         8+      DB      7
0008            8+      DEFB    8,9,,,,,,,
0001            8+      IF      1
0008 08         8+      DB      8
0009            8+      DEFB    9,,,,,,,,
0001            8+      IF      1
0009 09         8+      DB      9
000A            8+      DEFB    ,,,,,,,,
0000            8+      IF      0
                8+      ENDIF
                8+      ENDIF
                8+      ENDIF
                8+      ENDIF
                8+      ENDIF
                8+      ENDIF
                8+      ENDIF
                8+      ENDIF
                8+      ENDIF
                8+      ENDIF
                8+      ENDIF
                9 ;
```

16) New Error Messages

Each of the assembler's error messages were described in the M-ZAL SYSTEM MANUAL. This section describes those error messages that are new in M-ZAL RELEASE 2. In addition, there are now some new reasons why you might encounter some of the old error messages, these reasons are explained here also.

INVALID LABEL

This is an old error message. It will be produced if a label appears on an IF, ENDIF, or ENDM statement.

INVALID OPERAND(S)

This is an old error message. It will be produced if the operand of an IF statement is invalid or is missing. The IF statement will be ignored and assembly will continue normally. This will probably cause an "UNMATCHED ENDIF STATEMENT" error to occur later in the program.

UNMATCHED ENDIF STATEMENT

This new error message will be produced if an ENDIF statement is encountered when no IF statement is active. The statement will be ignored and assembly will continue normally.

MISSING ENDIF STATEMENT

This new error message will be produced if the end of a program is reached and an IF statement is still active. The assembly will be terminated normally.

UNDEFINED EXPRESSION ON IF STATEMENT

This new error message will be given if an IF statement is encountered with an undefined operand. This is a terminal error and will cause immediate termination of the assembly. It will occur during first pass, prior to the listing phase, but the assembler will list the offending statement and the statement that follows it to help you identify the source of the problem.

INVALID MACRO NAME

This new error message will be given if the label field on a MACRO statement is missing, or contains an invalid label, or contains the name MACRO, ENDM, IF, or ENDIF. The macro definition will be ignored.

INVALID MACRO DEFINITION IGNORED

This new error message is given if a MACRO statement is encountered after object code generation has begun. Object code generation begins as soon as a Z-80 opcode or a M-ZAL pseudo-opcode is encountered. The only statements that can appear before or in between macro definitions are assembler command statements (statements that begin with a *) and comment statements (statements that begin with a ;). This error will cause the offending MACRO statement to be ignored. Assembly will then continue normally.

EXTRA ENDM STATEMENT

This new error message will be given if an ENDM statement is encountered when a macro definition is not in progress. The statement will be ignored and assembly will continue.

INVALID PARAMETER

This new error message will be given if an invalid operand is detected on a macro invocation statement. Processing of any subsequent operands on this statement will be suspended and the macro expansion will begin. The invalid operand and any subsequent operands will be considered null during the macro expansion. Consider:

```
MAC3    A,B,'C D
```

In the above example a macro named MAC3 is being invoked. The third operand of the macro invocation is invalid because it does not contain a closing quote. This will cause the "INVALID PARAMETER" error message to be given. Expansion of the MAC3 macro will begin using a value of A for the first symbolic parameter, a value of B for the second symbolic parameter, and a null value for the third symbolic parameter.

INVALID DUMMY PARAMETER

This new error message will be given if a symbolic parameter on a MACRO statement is invalid. Parsing of the MACRO statement will be suspended at the offending parameter. Consider:

```
BADONE  MACRO   ?IX,?OK
```

The above example will cause this error message to be given because ?IX is not a valid symbolic parameter (because IX is not a valid M-ZAL label). Although processing of this macro definition will continue, parsing of the MACRO statement itself does not. This means that the ?OK symbolic parameter will not be recognized as such in the macro body.

Note that the special symbolic parameter names ?INDEX and ?PARM# are valid and can be specified on MACRO statements. If they are used in this fashion, however, they cannot be used for their special purpose as they will reflect the corresponding operand value during a macro expansion instead.

OUT OF MEMORY DURING MACRO DEFINITION

Each macro definition is compressed into a special format and stored in memory during the assembly process. This terminal error message will be given if your macro definitions are so large that they fill all available memory. The assembly will be terminated during the first pass, prior to the listing phase. The assembler will list the statement that was being processed when the error occurred as well as the statement that follows it to help you identify the location of the error.

OUT OF MEMORY DURING NESTED MACRO EXPANSION

As mentioned earlier, when macros are nested each individual macro expansion is given a unique set of values for symbolic parameters and the special parameters ?INDEX and ?PARM#. These unique values for each nested level of macro expansion are stored in a special buffer area in memory. This error message is given if this buffer overflows. This error can occur if macros are nested too deeply. There is no specific limit to how deeply macros can be nested, the practical limit depends upon how many parameters are being used and how long they are.

This error will cause immediate termination of the assembly. It will occur during first pass, prior to the listing phase, but the assembler will list the offending statement and the statement that follows it to help you identify the source of the problem.

V. NEW FEATURES IN LINKER

The LINKER program has been enhanced by the addition of three new commands in M-ZAL RELEASE 2. They are the DISPLAY, ZAP, and SPEED commands. The SPEED command is only available on the Model 3.

1) DISPLAY Command

The DISPLAY command gives you the ability to examine the contents of modules that have been loaded. Its format is as follows:

DISPLAY #### {#### {P}}

The first operand, which is always required, must be specified as a constant value using standard syntax. It specifies the starting address of the data to be examined. The second operand, which is optional, specifies how many bytes of data are to be examined. If omitted, a value of 4 is used. If the third operand is specified as "P" then the display will be routed to the printer instead of to the video screen.

This command causes the LINKER to display the requested data in hexadecimal. In addition, the name of the module containing the requested data is listed.

For example, the command:

DISPLAY 7000H 10

will cause the LINKER to display the name of the module which was loaded at address 7000H as well as the contents of that module from addresses 7000H through 7009H.

The command DISPLAY 7000H 10 P will perform the same operation except the output will be directed to the printer.

The command DISPLAY 7000H will perform the same function except only the four bytes from 7000H through 7003H will be listed.

The command verb DISPLAY can be abbreviated as DISP if desired.

Note that this command can only display data within a single module at a time. If you request a DISPLAY of more bytes than exist in the module, the extra bytes will be ignored. Furthermore, you cannot request a DISPLAY of an address that has not been loaded via the LOADA or LOADR commands, such a request will result in an "ADDRESS NOT FOUND" error.

2) ZAP Command

The ZAP command gives you the ability to modify the contents of modules that have been loaded. Its format is as follows:

ZAP #### hhhhhhhh....

The first operand specifies the address to be modified and is specified as a constant using the standard assembler language syntax. The second operand specifies the new data to be placed at the first operand address. The second operand must be specified in hexadecimal, with no trailing "H" character. One or more bytes may be specified in the second operand. No leading zero should be used if the first nibble of the second operand is A through F.

For example, the command:

ZAP 7000H EDB0

will cause the contents of location 7000H to be changed to 0EDH and the contents of location 7001H to be changed to 0B0H.

This command can only be used to alter data at locations that have been previously loaded via LOADA or LOADR commands. The LINKER will indicate the name of the module containing the data that has been altered when this command is entered.

3) SPEED Command

The SPEED command has been provided to allow Model 3 users to dynamically switch between low and high speed cassette operations while using the LINKER. It has no operands.

When the SPEED command is entered, the LINKER will call the ROM routine that prompts the user with the "Cass?" question. Reply with either L for low speed (500 baud) or H for high speed (1500 baud) cassette operations. This command will have no effect if used on a Model 1.

4) New LINKER Error Messages

INVALID COMMAND

This is not a new error message, but it can now be given for a new reason. Specifically, if the second operand of a ZAP command is not in the format of a string of hexadecimal characaters, this error will be given.

ADDRESS NOT FOUND

This message will be given if the first operand of a DISPLAY or ZAP command specifies an address that is not known to the LINKER (i.e.: not contained within any of the presently loaded modules).

ZAP DATA EXCEEDS EXTENT

A single ZAP command can only affect data contained within a single extent of a module. This error is given if the second operand of a ZAP command specifies more bytes than exist from the starting ZAP address to the end of the extent. In this case no data will be zapped. For example, consider:

```
+LOADA   TEMP
+MAP
                                MODULE MAP
                                ------ ---
 MODULE NAME  ADDRESSES  ENTRIES              EXTRNS
 -----------  ---------  -----------------    -----------
 TEMP         7000-7100
              7200-7FFF
 -----------  ---------  -----------------    -----------

+ZAP     7000H   010203
 ZAP MADE IN MODULE TEMP
+ZAP     7100H   FF
 ZAP MADE IN MODULE TEMP
+ZAP     7100H   FF00
 ZAP DATA EXCEEDS EXTENT
```

The last ZAP command shown above produced an error message because it attempted to alter data outside the extent from 7000H to 7100H.